UNIVERSITÀ DI PARMA
Dipartimento di Ingegneria e Architettura

Software vulnerabilities

Luca Veltri

(mail.to: luca.veltri@unipr.it)

Course of Cybersecurity, 2022/2023

http://www.tlc.unipr.it/veltri

Software vulnerabilities

- Vulnerabilities in application, utility, or operating system code
- Software vulnerabilities are often caused by a bug or weakness present in the software
  - **application implementations**
  - **operating systems**
- Many computer security vulnerabilities result from poor programming practices

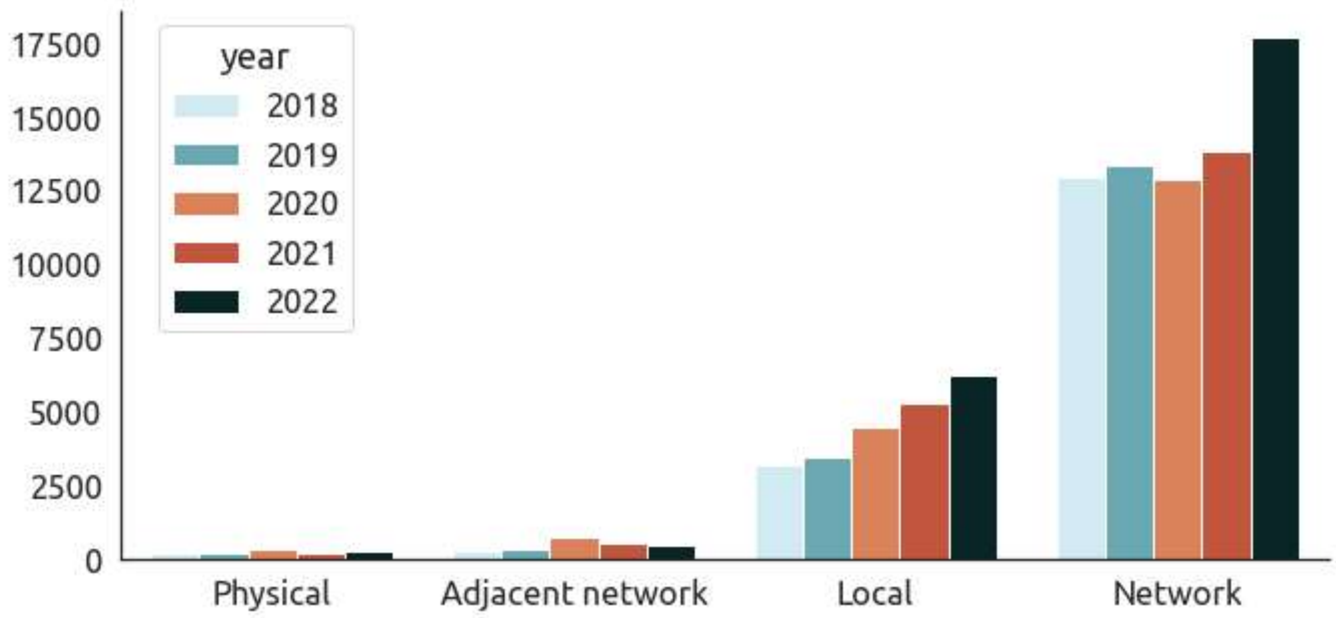
2022 CWE Most Dangerous Software Weaknesses

1	CWE-787	Out-of-bounds Write
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
4	CWE-20	Improper Input Validation
5	CWE-125	Out-of-bounds Read
6	CWE-78	Improper neutralization of special elements used in an OS command (OS Command Injection'
7	CWE-416	Use After Free
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
9	CWE-352	Cross-Site Request Forgery (CSRF)
10	CWE-434	Unrestricted Upload of File with Dangerous Type
11	CWE-476	NULL Pointer Dereference
12	CWE-502	Deserialization of Untrusted Data
13	CWE-190	Integer Overflow or Wraparound
14	CWE-287	Improper Authentication
15	CWE-798	Use of Hard-coded Credentials
16	CWE-862	Missing Authorization
17	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')
18	CWE-306	Missing Authentication for Critical Function
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer
20	CWE-276	Incorrect Default Permissions
21	CWE-918	Server-Side Request Forgery (SSRF)
22	CWE-362	Concurrent execution using shared resource with improper synchronization (Race condition)
23	CWE-400	Uncontrolled Resource Consumption
24	CWE-611	Improper Restriction of XML External Entity Reference
25	CWE-94	Improper Control of Generation of Code ('Code Injection')

https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

CVE analysis

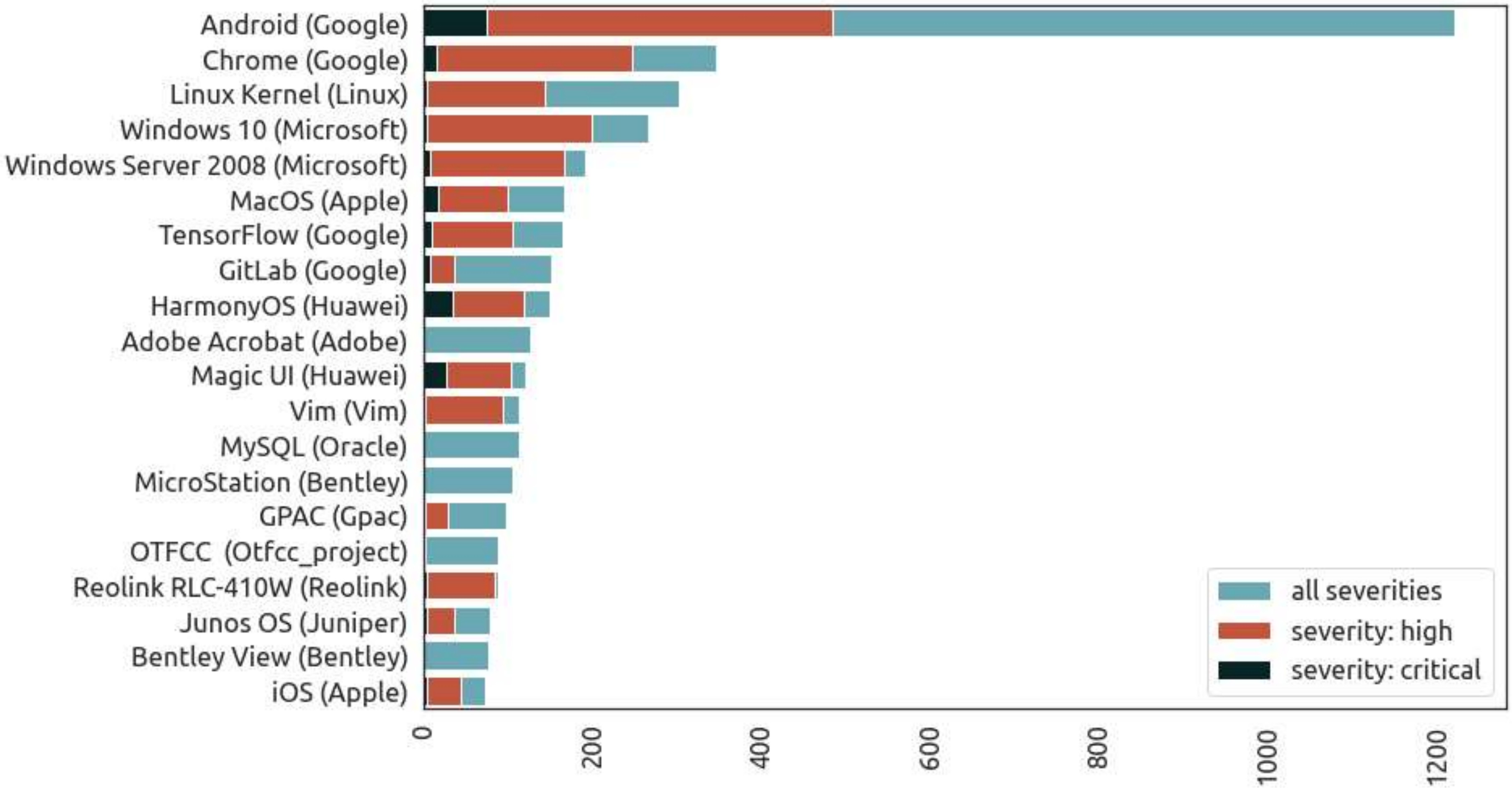
- Attack vectors(*):



(*) https://thestack.technology

CVE analysis (cont.)

- The top 20 products with the most CVEs in 2022:



(*) https://thestack.technology

Top 10 web application vulnerabilities (2021)

- Open Web Application Security Project (OWASP) Top 10 vulnerabilities
  1) **Broken Access Control**
  2) **Cryptographic Failures**
  3) **Injection**
  4) **Insecure Design**
  5) **Security Misconfiguration**
  6) **Vulnerable and Outdated Components**
  7) **Identification and Authentication Failures**
  8) **Software and Data Integrity Failures**
  9) **Security Logging and Monitoring Failures**
  10) **Server Side Request Forgery (SSRF)**

https://owasp.org/www-project-top-ten/
https://www.veracode.com/security/owasp-top-10

Software Error Categories

- Possible classification CWE/SANS Most Dangerous Software Errors (2011):
  - **1) Insecure Interaction Between Components**
    - Improper Neutralization of Special Elements used in an SQL Command (“SQL Injection”)
    - Improper Neutralization of Special Elements used in an OS Command (“OS Command Injection”)
    - Improper Neutralization of Input During Web Page Generation (“Cross-site Scripting”)
    - Unrestricted Upload of File with Dangerous Type
    - Cross-Site Request Forgery (CSRF)
    - URL Redirection to Untrusted Site (“Open Redirect”)

Software Error Categories (cont.)

- Classification (cont.):
  - **2) Risky Resource Management**
    - Buffer Copy without Checking Size of Input ("Classic Buffer Overflow")
    - Improper Limitation of a Pathname to a Restricted Directory ("Path Traversal")
    - Download of Code Without Integrity Check
    - Inclusion of Functionality from Untrusted Control Sphere
    - Use of Potentially Dangerous Function
    - Incorrect Calculation of Buffer Size
    - Uncontrolled Format String
    - Integer Overflow or Wraparound

Software Error Categories (cont.)

- Classification (cont.):
  - **3) Porous Defenses**
    - Missing Authentication for Critical Function
    - Missing Authorization
    - Use of Hard-coded Credentials
    - Missing Encryption of Sensitive Data
    - Reliance on Untrusted Inputs in a Security Decision
    - Execution with Unnecessary Privileges
    - Incorrect Authorization
    - Incorrect Permission Assignment for Critical Resource
    - Use of a Broken or Risky Cryptographic Algorithm
    - Improper Restriction of Excessive Authentication Attempts
    - Use of a One-Way Hash without a Salt

Software Error Categories (cont.)

- While type 3 (Porous Defense) vulnerabilities are mainly related to poor implementation of protection mechanisms, type 1 and 2 vulnerabilities mainly refer to errors/weaknesses due to improper handling of program inputs
- These can mainly due to two different levels:
  - **errors/weaknesses at programming language level**
    - how programs are written and how they are translated into machine code
      - issues often related on the size of input data
        - » e.g. buffer overflow
  - **errors/weaknesses at program logic level**
    - how inputs are interpreted and handled by the logic of the program
    - incorrect handle of remote user inputs that makes the program to perform other actions
      - often due to execution of action based on user input values without a proper check of correctness (validation) of the input
        - » Injection attacks

Buffer Overflow

- Also known as a buffer overrun
- Very common attack mechanism
  - **First widely used by the Morris Worm in 1988**
- Prevention techniques known
- Still of major concern
  - **Legacy of buggy code in widely deployed operating systems and applications**
  - **Continued careless programming practices by programmers**
- Defined in the NIST Glossary of Key Information Security Terms as follows:
  - **A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system**

Buffer Overflow Basics

- Programming error when a process attempts to store data beyond the limits of a fixed-sized buffer
- Overwrites adjacent memory locations
  - **Locations could hold other program variables, parameters, or program control flow data**
- Buffer could be located on the stack, in the heap, or in the data section of the process
- Consequences:
  - **Corruption of program data**
  - **Unexpected transfer of control**
  - **Memory access violations**
  - **Execution of code chosen by attacker**

Buffer overflow example

- Simple example of buffer overflow C code

```
#define FALSE 0
#define TRUE 1

void get_password(char* dest) { // get a password from a DB
     strcpy(dest,"SECRET");
}

int main(int argc, char *argv[]) {
     int valid= FALSE;
     char str1[8];
     char str2[8];
     printf("insert valid password: ");
     get_password(str1); // read the password from the user
     gets(str2);
     if (strncmp(str1,str2,8)==0) valid= TRUE; else valid= FALSE;
     printf("str1: %s\nstr2: %s\nvalid: %d\n", str1, str2,
valid);
}
```

Buffer overflow example (cont.)

```
insert password: test
str1: SECRET
str2: test
valid: 0

insert password: 123456789abcdefghijklmno
str1: 9abcdefg
str2: 123456789abcdefg
valid: 0

insert password: aaaabbbbaaaabbbbccccdddd
str1: aaaabbbb☺
str2: aaaabbbbaaaabbbb☺
valid: 1

insert password: badinputbadinput
str1: badinput☺
str2: badinputbadinput☺
valid: 1
```

Buffer overflow example (cont.)

Memory Address	Before gets(str2)	After gets(str2)	Contains value of
. . . .	. . . .	. . . .	
bffffbf4	34fcffbf 4 . . .	34fcffbf 3 . . .	argv
bffffbf0	01000000	01000000	argc
bffffbec	c6bd0340 . . . @	c6bd0340 . . . @	return addr
bffffbe8	08fcffbf	08fcffbf	old base ptr
bffffbe4	00000000	01000000	valid
bffffbe0	80640140 . d . @	00640140 . d . @	
bffffbdc	54001540 T . . @	4e505554 N P U T	str1[4-7]
bffffbd8	53544152 S T A R	42414449 B A D I	str1[0-3]
bffffbd4	00850408	4e505554 N P U T	str2[4-7]
bffffbd0	30561540 0 V . @	42414449 B A D I	str2[0-3]
. . . .	. . . .	. . . .	

Buffer overflow attacks

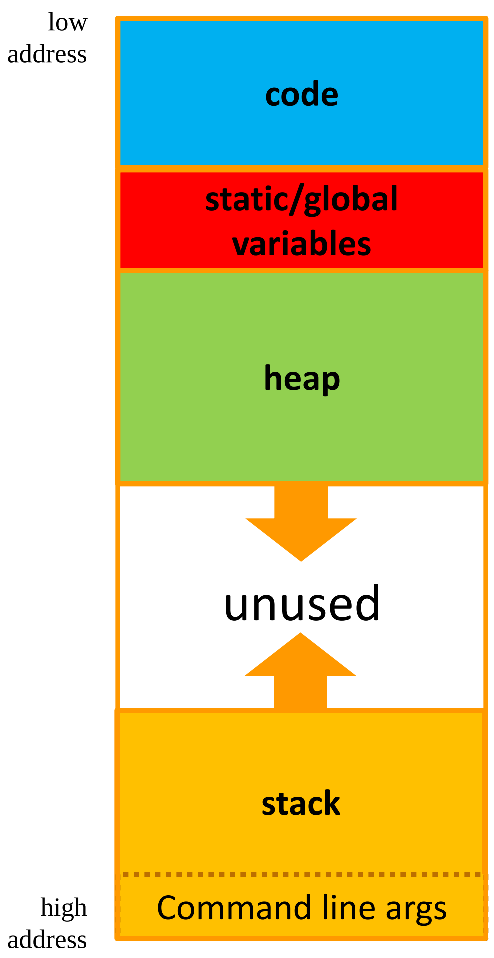
- To exploit a buffer overflow an attacker needs:
  - **To identify a buffer overflow vulnerability in some program that can be triggered using externally sourced data under the attacker's control**
  - **To understand how that buffer is stored in memory and determine potential for corruption**
- Identifying vulnerable programs can be done by:
  - **Inspection of program source**
  - **Tracing the execution of programs as they process oversized input**
  - **Using tools such as fuzzing to automatically identify potentially vulnerable programs**
- What the attacker does with the resulting corruption of memory varies considerably, depending on what values are being overwritten

Stack Buffer Overflows

- Occur when buffer is located on stack
  - **also referred to as stack smashing**
  - **used by Morris Worm**
  - **exploits include an unchecked buffer overflow**
- Are still being widely exploited
- Stack frame
  - **When one function calls another it needs somewhere to save the return address**
  - **Also needs locations to save the parameters to be passed in to the called function and to possibly save register values**

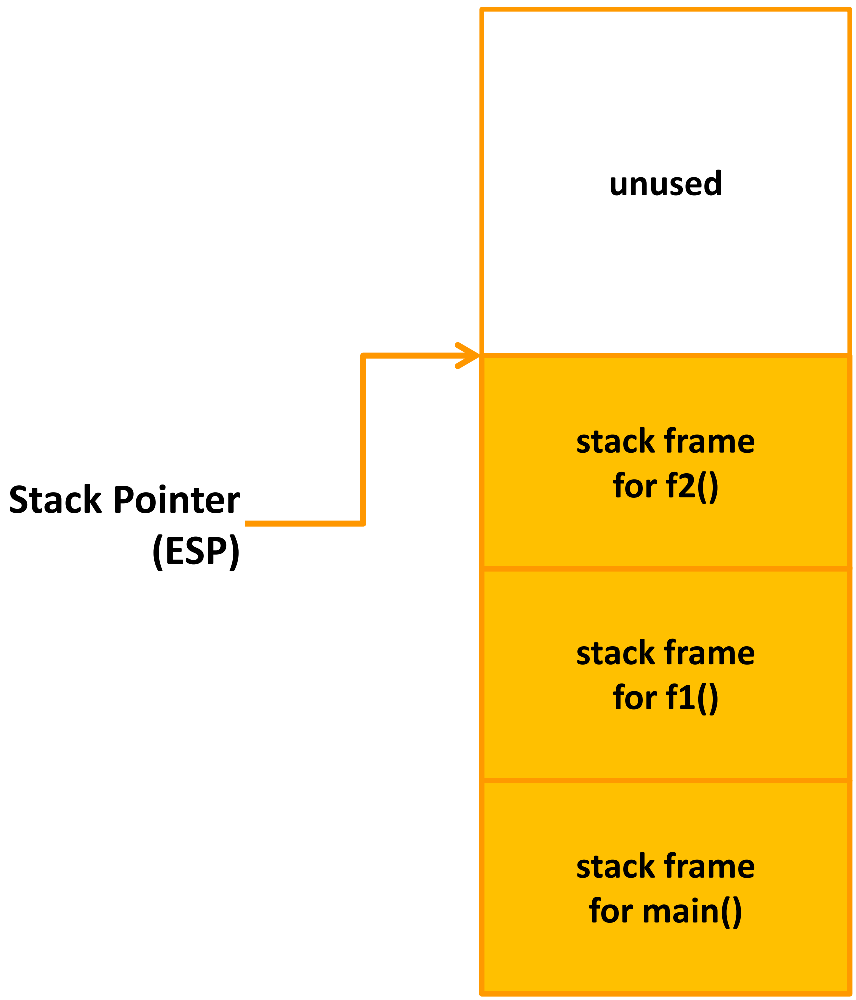
Program memory allocation

- Memory is allocated for each process (a running program) to store data and code
- This allocated memory consists of different segments:
  - **stack: for local variables**
  - **heap: for dynamic memory**
  - **data segment:**
    - global uninitialized variables
    - global initialized variables
  - **code segment**



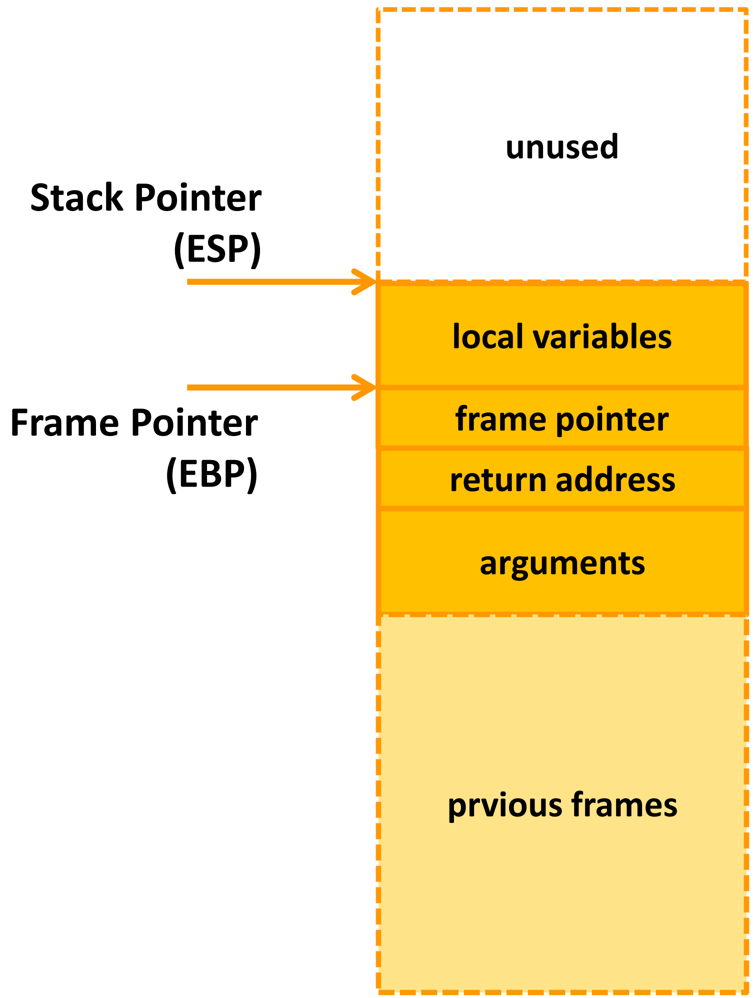
The stack

- The precise structure and organization of the stack depends on system architecture, operating system, and compilers we are using
- Typically, the stack grows downward
- The stack pointer (SP) refers to the last element on the stack
  - **on x86 architectures, the stack pointer is stored in the ESP (Extended Stack Pointer) register**



x86 stack frame

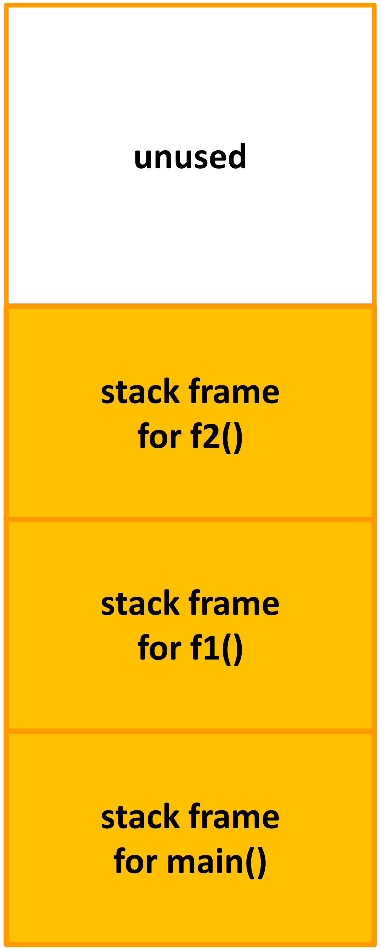
- In x86 architecture, each stack frame contains:
  - **Function arguments**
  - **Local variables**
  - **Copies of registries that must be restored:**
    - return address
    - previous frame pointer
- Frame pointer, named Extended Base Pointer (EBP), provides a starting point to local variables



The stack (cont.)

- The stack consists of a sequence of *stack frames* (or activation records), each for each function call:
  - **allocated on *call***
  - **de-allocated on *return***

- In the stack example:
  - **main() called f1()**
  - **f1() called f2()**
  - **f1() and f2() can be also the same function**
    - in case of recursion

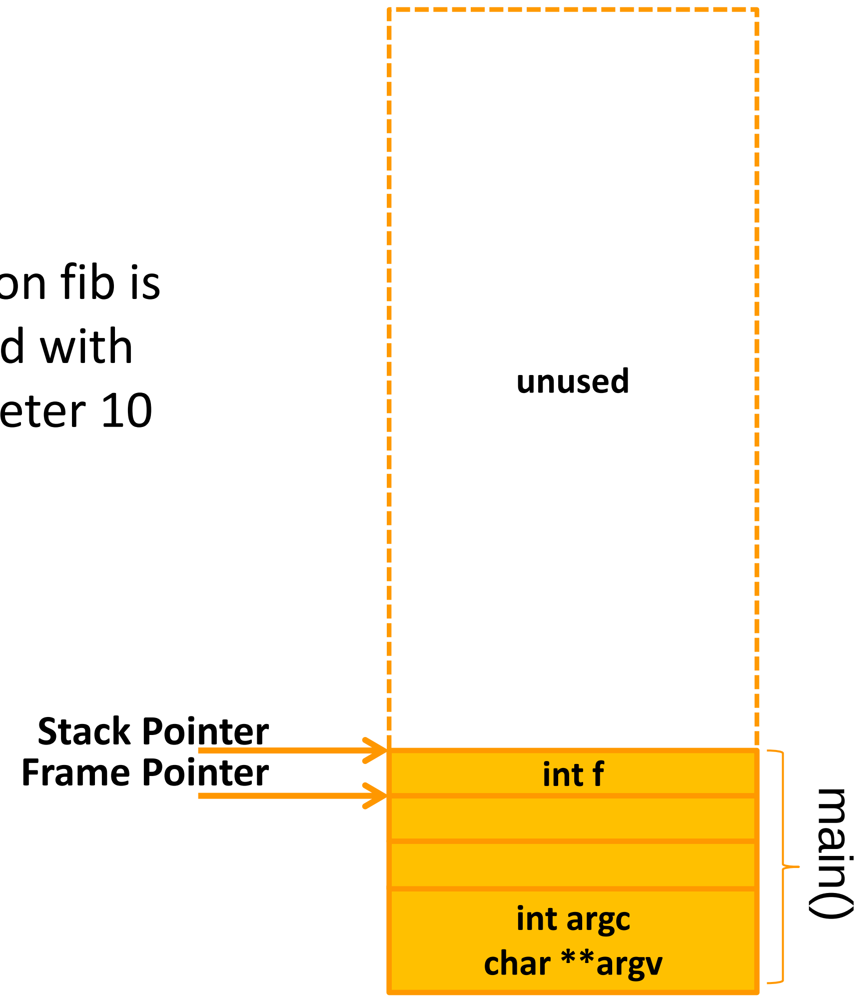


Stack frame: example

```
#include<cstdio>

int fibonacci(int n) {
    int f1, f2;
    if (n<=2) return 1;
    else {
        f1= fibonacci(n-1);
        f2= fibonacci(n-2);
        return f1+f2;
    }
}
int main () {
    int f= fibonacci(10);
    printf("fib(10)= %d\n",f);
}
```

Function fib is invoked with parameter 10

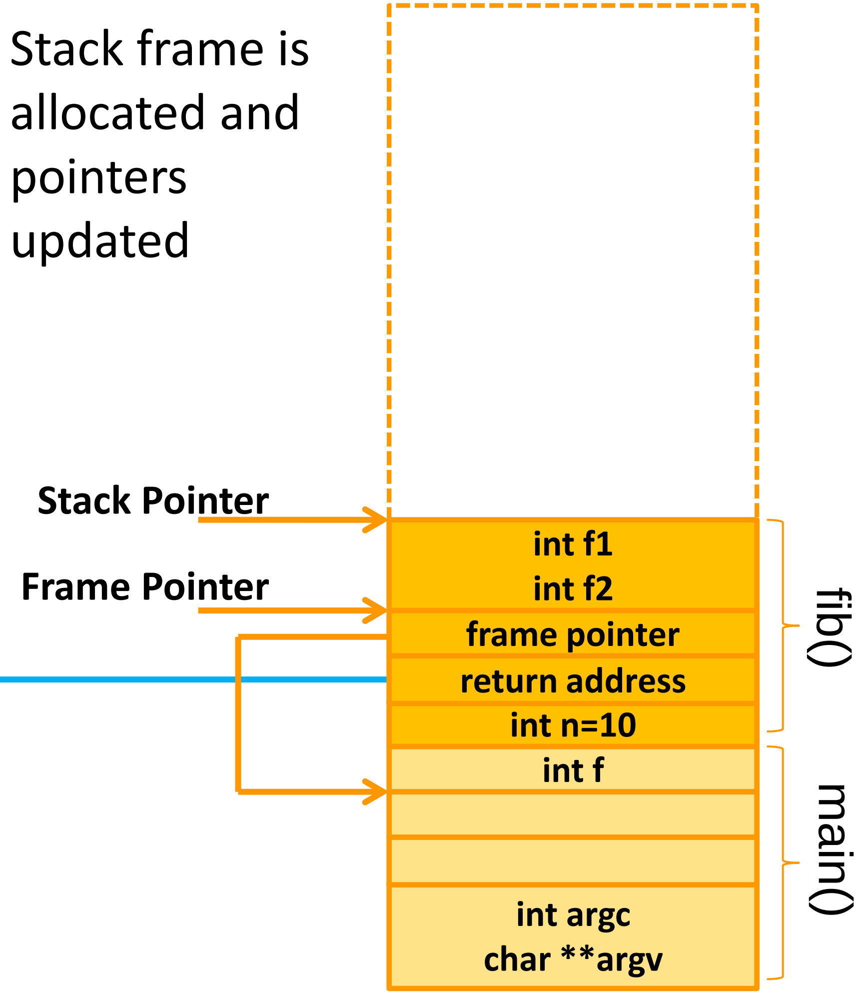


Stack frame: example

```
#include<cstdio>

int fibonacci(int n) {
    int f1, f2;
    if (n<=2) return 1;
    else {
        f1= fibonacci(n-1);
        f2= fibonacci(n-2);
        return f1+f2;
    }
}
int main () {
    int f= fibonacci(10);
    printf("fib(10)= %d\n",f);
}
```

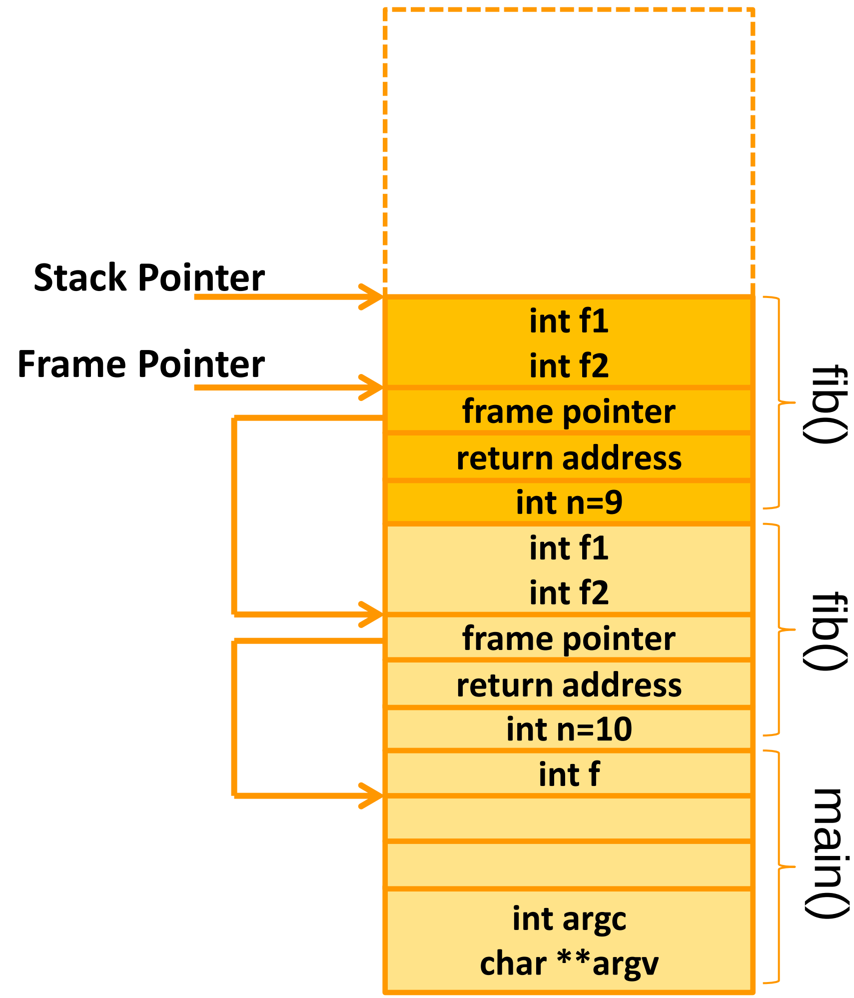
Stack frame is allocated and pointers updated



Stack frame: example

```
#include<cstdio>

int fibonacci(int n) {
    int f1, f2;
    if (n<=2) return 1;
    else {
        f1= fibonacci(n-1);
        f2= fibonacci(n-2);
        return f1+f2;
    }
}
int main () {
    int f= fibonacci(10);
    printf("fib(10)= %d\n",f);
}
```



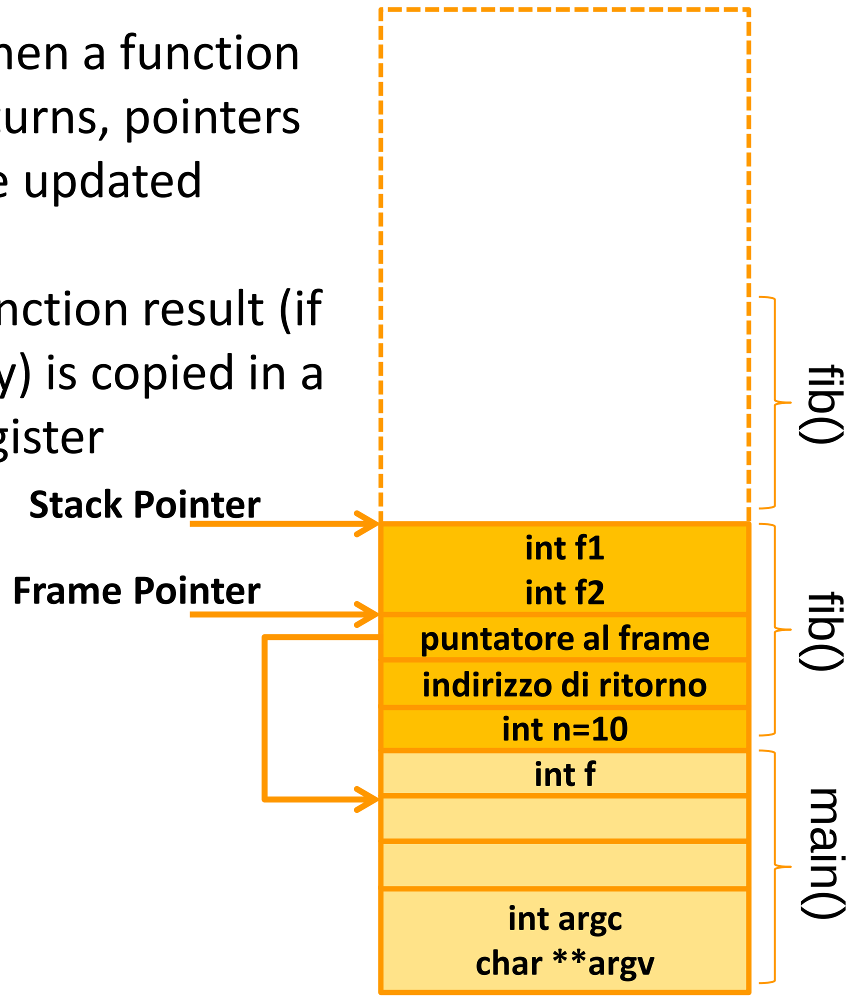
Stack frame: example

```
#include<cstdio>

int fibonacci(int n) {
    int f1, f2;
    if (n<=2) return 1;
    else {
        f1= fibonacci(n-1);
        f2= fibonacci(n-2);
        return f1+f2;
    }
}
int main () {
    int f= fibonacci(10);
    printf("fib(10)= %d\n",f);
}
```

When a function returns, pointers are updated

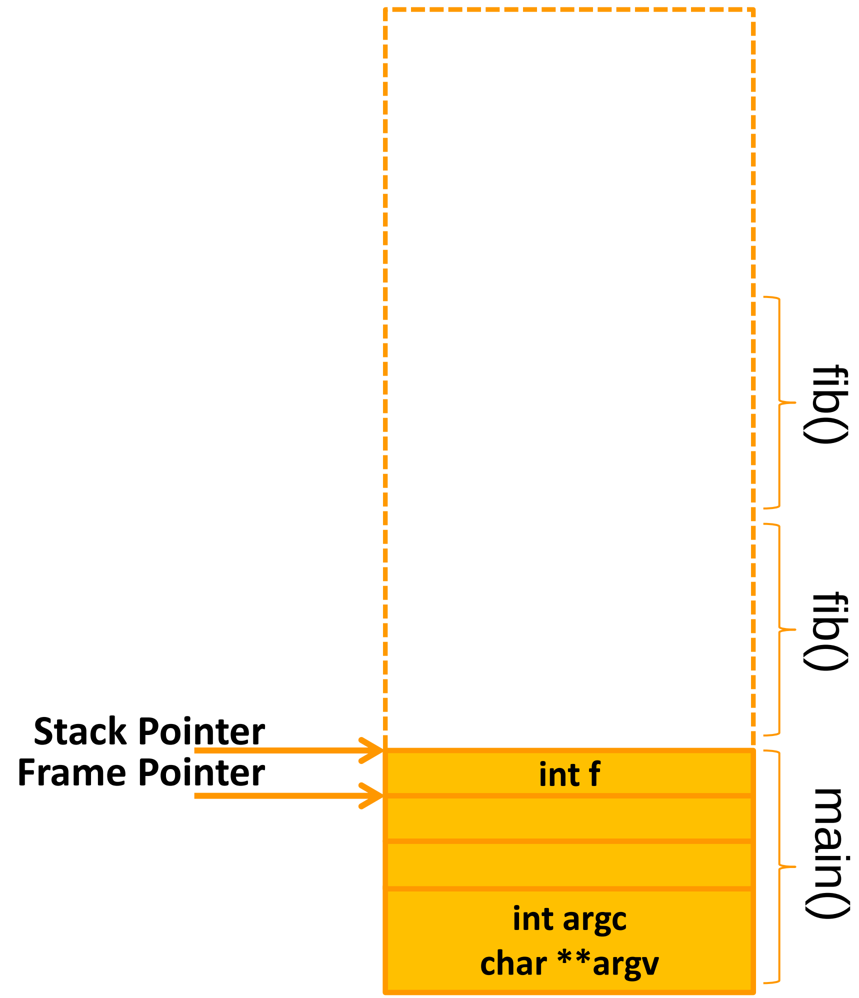
Function result (if any) is copied in a register



Stack frame: example

```
#include<cstdio>

int fibonacci(int n) {
    int f1, f2;
    if (n<=2) return 1;
    else {
        f1= fibonacci(n-1);
        f2= fibonacci(n-2);
        return f1+f2;
    }
}
int main () {
    int f= fibonacci(10);
    printf("fib(10)= %d\n",f);
}
```



The heap

- Memory allocation and de-allocation in the stack is very fast
  - **However, this memory cannot be used after a function returns**
- The heap is used to store dynamically allocated data that outlive function calls:
  - **This area is under programmer's responsibility**

Simple Stack Overflow Example

```
void hello(char *tag)
{
    char inp[16];

    printf("Enter value for %s: ", tag);
    gets(inp);
    printf("Hello your %s is %s\n", tag, inp);
}
```

(a) Basic stack overflow C code

```
$ cc -g -o buffer2 buffer2.c

$ ./buffer2
Enter value for name: Bill and Lawrie
Hello your name is Bill and Lawrie
buffer2 done

$ ./buffer2
Enter value for name: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Segmentation fault (core dumped)

$ perl -e 'print pack("H*", "414243444546474851525354555657586162636465666768
e8ffffbf948304080a4e4e4e4e0a");' | ./buffer2
Enter value for name:
Hello your Re?pyy]uEA is ABCDEFGHQRSTUVWXabcdefguyu
Enter value for Kyyu:
Hello your Kyyu is NNNN
Segmentation fault (core dumped)
```

(b) Basic stack overflow example runs

Simple Stack Overflow Stack Values

Memory Address	Before gets(inp)	After gets(inp)	Contains value of
. . . .	. . . .	. . . .	
bffffbe0	3e850408 > . . .	00850408	tag
bffffbdc	f0830408	94830408	return addr
bffffbd8	e8fbffbf	e8ffffbf	old base ptr
bffffbd4	60840408 ` . . .	65666768 e f g h	
bffffbd0	30561540 0 V . @	61626364 a b c d	
bffffbcc	1b840408	55565758 U V W X	inp[12-15]
bffffbc8	e8fbffbf	51525354 Q R S T	inp[8-11]
bffffbc4	3cfcffbf < . . .	45464748 E F G H	inp[4-7]
bffffbc0	34fcffbf 4 . . .	41424344 A B C D	inp[0-3]
. . . .	. . . .	. . . .	

Shellcode

- Code supplied by attacker
  - **Often saved in buffer being overflowed**
  - **Traditionally transferred control to a user command-line interpreter (shell)**
- Machine code
  - **Specific to processor and operating system**
  - **Traditionally needed good assembly language skills to create**
  - **More recently a number of sites and tools have been developed that automate this process**
    - e.g. Metasploit Project - it provides useful information to people who perform penetration, IDS signature development, and exploit research

Example UNIX Shellcode

```
int main (int argc, char *argv[])
{
    char *sh;
    char *args[2];


    sh = "/bin/sh";
    args[0] = sh;
    args[1] = NULL;
    execve (sh, args, NULL);
}
```

(a) Desired shellcode code in C

```
      nop
      nop                       //end of nop sled
      jmp find                  //jump to end of code
cont: pop %esi                  //pop address of sh off stack into %esi
      xor %eax, %eax            //zero contents of EAX
      mov %al, 0x7(%esi)        //copy zero byte to end of string sh (%esi)
      lea (%esi), %ebx          //load address of sh (%esi) into %ebx
      mov %ebx,0x8(%esi)        //save address of sh in args [0] (%esi+8)
      mov %eax,0xc(%esi)        //copy zero to args[1] (%esi+c)
      mov $0xb,%al              //copy execve syscall number (11) to AL
      mov %esi,%ebx             //copy address of sh (%esi) into %ebx
      lea 0x8(%esi),%ecx        //copy address of args (%esi+8) to %ecx
      lea 0xc(%esi),%edx        //copy address of args[1] (%esi+c) to %edx
      int $0x80                 //software interrupt to execute syscall
find: call cont                 //call cont which saves next address on stack
sh:   .string "/bin/sh"         //string constant
args: .long 0                   //space used for args array
      .long 0                   //args[1] and also NULL for env array
```

(b) Equivalent position-independent x86 assembly code

```
90 90 eb 1a 5e 31 c0 88 46 07 8d 1e 89 5e 08 89
46 0c b0 0b 89 f3 8d 4e 08 8d 56 0c cd 80 e8 e1
ff ff ff 2f 62 69 6e 2f 73 68 20 20 20 20 20 20
```

(c) Hexadecimal values for compiled x86 machine code

Buffer Overflow Defenses

- Buffer overflows are widely exploited

- Two broad defense approaches
  - **Compile-time**
    - Aim to harden programs to resist attacks in new programs
  - **Run-time**
    - Aim to detect and abort attacks in existing programs

Compile-Time Defenses: Programming Language

- Use a modern high-level language
  - **Not vulnerable to buffer overflow attacks**
  - **Compiler enforces range checks and permissible operations on variables**
- Disadvantages
  - **Additional code must be executed at run time to impose checks**
  - **Flexibility and safety comes at a cost in resource use**
  - **Distance from the underlying machine language and architecture means that access to some instructions and hardware resources is lost**
  - **Limits their usefulness in writing code, such as device drivers, that must interact with such resources**

Compile-Time Defenses: Safe Coding Techniques

- C designers placed much more emphasis on space efficiency and performance considerations than on type safety
  - **Assumed programmers would exercise due care in writing code**

- Programmers need to inspect the code and rewrite any unsafe coding
  - **An example of this is the Open BSD project**

- Programmers have audited the existing code base, including the operating system, standard libraries, and common utilities
  - **This has resulted in what is widely regarded as one of the safest operating systems in widespread use**

Examples of Unsafe C Code

```
int copy_buf(char *to, int pos, char *from, int len)
{
    int i;
    for (i=0; i<len; i++) {
        to[pos] = from[i];
        pos++;
    }
    return pos;
}
```

(a) Unsafe byte copy

```
short read_chunk(FILE fil, char *to)
{
    short len;
    fread(&len, 2, 1, fil);         /* read length of binary data */
    fread(to, 1, len, fil);         /* read len bytes of binary data
    return len;
}
```

(b) Unsafe byte input

Compile-Time Defenses: Language Extensions/Safe Libraries

- Handling dynamically allocated memory is more problematic because the size information is not available at compile time
  - **Requires an extension and the use of library routines**
    - Programs and libraries need to be recompiled
    - Likely to have problems with third-party applications

- Concern with C is use of unsafe standard library routines
  - **One approach has been to replace these with safer variants**
    - Libsafe is an example
    - Library is implemented as a dynamic library arranged to load before the existing standard libraries

Compile-Time Defenses: Stack Protection

- Add function entry and exit code to check stack for signs of corruption
- Use random canary
  - **value needs to be unpredictable**
- Stackshield and Return Address Defender (RAD)
  - **GCC extensions that include additional function entry and exit code**
    - function entry writes a copy of the return address to a safe region of memory
    - function exit code checks the return address in the stack frame against the saved copy
    - if change is found, aborts the program

Run-Time Defenses: Executable Address Space Protection

- Use virtual memory support to make some regions of memory non-executable
  - **Requires support from memory management unit (MMU)**
  - **Long existed on SPARC / Solaris systems**
  - **Recent on x86 Linux/Unix/Windows systems**

- Issues
  - **Support for executable stack code**
  - **Special provisions are needed**

Run-Time Defenses: Address Space Randomization

- Manipulate location of key data structures
  - **stack, heap, global data**
  - **using random shift for each process**
  - **large address range on modern systems means wasting some has negligible impact**

- Randomize location of heap buffers

- Random location of standard library functions

Replacement Stack Frame

- Variant that overwrites buffer and saved frame pointer address
  - **Saved frame pointer value is changed to refer to a dummy stack frame**
  - **Current function returns to the replacement dummy frame**
  - **Control is transferred to the shellcode in the overwritten buffer**
- Off-by-one attacks
  - **Coding error that allows one more byte to be copied than there is space available**
- Defenses
  - **Any stack protection mechanisms to detect modifications to the stack frame or return address by function exit code**
  - **Use non-executable stacks**
  - **Randomization of the stack in memory and of system libraries**

Return to System Call

- Stack overflow variant replaces return address with standard library function
  - **Response to non-executable stack defenses**
  - **Attacker constructs suitable parameters on stack above return address**
  - **Function returns and library function executes**
  - **Attacker may need exact buffer address**
  - **Can even chain two library calls**
- Defenses
  - **Any stack protection mechanisms to detect modifications to the stack frame or return address by function exit code**
  - **Use non-executable stacks**
  - **Randomization of the stack in memory and of system libraries**

Heap Overflow

- Attack buffer located in heap
  - **Typically located above program code**
  - **Memory is requested by programs to use in dynamic data structures (such as linked lists of records)**
- No return address
  - **Hence no easy transfer of control**
  - **May have function pointers can exploit**
  - **Or manipulate management data structures**
- Defenses
  - **Making the heap non-executable**
  - **Randomizing the allocation of memory on the heap**

Example Heap Overflow Attack

```
/* record type to allocate on heap */
typedef struct chunk {
    char inp[64];               /* vulnerable input buffer */
    void (*process)(char *);    /* pointer to function to process inp */
} chunk_t;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    chunk_t *next;

    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

(a) Vulnerable heap overflow C code

Global Data Overflow

- Defenses
  - **Non executable or random global data region**
  - **Move function pointers**
  - **Guard pages**

- Can attack buffer located in global data
  - **May be located above program code**
  - **If has function pointer and vulnerable buffer**
  - **Or adjacent process management tables**
  - **Aim to overwrite function pointer later called**

Example Global Data Overflow Attack

```
/* global static data - will be targeted for attack */
struct chunk {
    char inp[64];          /* input buffer */
    void (*process)(char *); /* pointer to function to process it */
} chunk;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer6 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    setbuf(stdin, NULL);
    chunk.process = showlen;
    printf("Enter value: ");
    gets(chunk.inp);
    chunk.process(chunk.inp);
    printf("buffer6 done\n");
}
```

(a) Vulnerable global data overflow C code

Injection Attacks

- Flaws relating to invalid handling of input data, specifically when program input data can accidentally or deliberately influence the flow of execution of the program
- Most common case when input data are passed as a parameter to another program on the system
- Most often occur in scripting languages
  - **e.g. Perl, PHP, Python, sh, SQL**
  - **often used as Web CGI scripts**

A Web CGI Injection Attack

```
#!/usr/bin/perl
# finger.cgi - finger CGI script using Perl5 CGI module

use CGI;
use CGI::Carp qw(fatalsToBrowser);
$q = new CGI; # create query object

# display HTML header
print $q->header,
$q->start_html('Finger User'),
$q->h1('Finger User');
print "<pre>";

# get name of user and display their finger details
$user = $q->param("user");
print `/usr/bin/finger -sh $user`;

# display HTML footer
print "</pre>";
print $q->end_html;
```

(a) Unsafe Perl finger CGI script

```
<html><head><title>Finger User</title></head><body>
<h1>Finger User</h1>
<form method=post action="finger.cgi">
<b>Username to finger</b>: <input type=text name=user value="">
<p><input type=submit value="Finger User">
</form></body></html>
```

(b) Finger form

Example of correct input:

`lpb`

Example of command injection:

`xxx; echo attack success; ls -lfinger*`

```
Finger User
Login Name     TTY Idle Login Time Where
lpb Lawrie Brown   p0 Sat 15:24 ppp41.grapevine
Finger User
attack success
-rwxr-xr-x 1 lpb staff 537 Oct 21 16:19 finger.cgi
-rw-r--r-- 1 lpb staff 251 Oct 21 16:14 finger.html
```

(c) Expected and subverted finger CGI responses

```
# get name of user and display their finger details
$user = $q->param("user");
die "The specified user contains illegal characters!"
unless ($user =~ /^\w+$/);
print `/usr/bin/finger -sh $user`;
```

(d) Safety extension to Perl finger CGI script

PHP/SQL Injection Example

```
$name = $_REQUEST['name'];
$query = "SELECT * FROM suppliers WHERE name = '" . $name . "';";
$result = mysql_query($query);
```

(a) Vulnerable PHP code

```
$name = $_REQUEST['name'];
$query = "SELECT * FROM suppliers WHERE name = '" .
mysql_real_escape_string($name) . "';";
$result = mysql_query($query);
```

(b) Safer PHP code

Example of injection input:

```
Bob'; drop table suppliers
```

Cross Site Scripting (XSS) Attacks

- Attacks where input provided by one user is subsequently output to another user
- Commonly seen in scripted Web applications
  - **vulnerability involves the inclusion of script code in the HTML content**
  - **script code may need to access data associated with other pages**
  - **browsers impose security checks and restrict data access to pages originating from the same site**
- Exploit assumption that all content from one site is equally trusted and hence is permitted to interact with other content from the site
- XSS reflection vulnerability
  - **attacker includes the malicious script content in data supplied to a site**

Input Fuzzing

- Developed by Professor Barton Miller at the University of Wisconsin Madison in 1989
- Software testing technique that uses randomly generated data as inputs to a program
  - **range of inputs is very large**
  - **intent is to determine if the program or function correctly handles abnormal inputs**
  - **simple, free of assumptions, cheap**
  - **assists with reliability as well as security**
- Can also use templates to generate classes of known problem inputs
  - **disadvantage is that bugs triggered by other forms of input would be missed**
  - **combination of approaches is needed for reasonably comprehensive coverage of the inputs**